CS1112 Spring 2015 Project 5 due Thursday 4/16 at 11pm

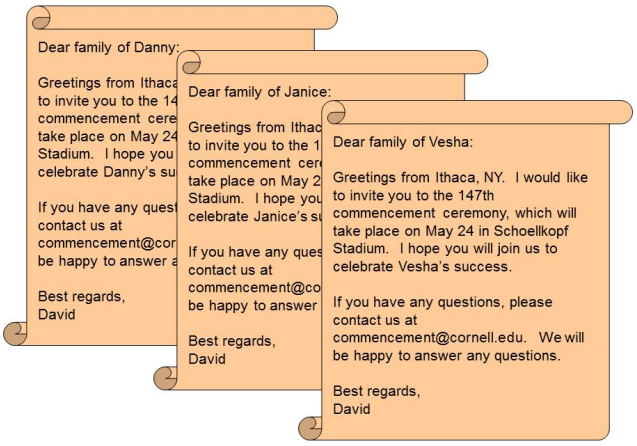
You must work either on your own or with one partner. If you work with a partner you must first register as a group in CMS and then submit your work as a group. *Adhere to the Code of Academic Integrity.* For a group, "you" below refers to "your group." You may discuss background issues and general strategies with others and seek help from the course staff, but the work that you submit must be your own. In particular, you may discuss general ideas with others but you may not work out the detailed solutions with others. It is not OK for you to see or hear another student's code and it is certainly not OK to copy code from another person or from published/Internet sources. If you feel that you cannot complete the assignment on you own, seek help from the course staff.

Objectives

Completing this project will solidify your understanding of character arrays, cell arrays, structures, and structure arrays. You will also work with text data files. You will test your function and subfunctions throughout program development.

1 May MATLAB Mail Merge?

Mail merge is a software functionality describing the production of multiple documents from a single template and a structured data source. One particular use of mail merge is to create personalized letters and address labels. The template contains fixed text—which will be the same in each output document—and placeholders—which are replaced by text from the data source.[1]



In this project you will complete a MATLAB function `mailMerge` to perform a mail merge from one sender to multiple recipients. The sender data, recipient data, as well as the letter template, are given in plain text files and you can use them for testing. You will submit only one file, `mailMerge.m`.

To make sure that you spend time on developing your program rather than waste time on looking up built-in functions, we list here the only three *string* handling built-in functions that you are allowed in this project: `strcmp`, `str2double`, and `strfind`. Function `strfind` is explained below and the other two are explained in lecture and *Insight*. You can of course still use built-in functions that are general, such as `length` which works with 1-d arrays including strings.

1.1 Get Ready for Testing

Start by downloading all the files (`mailMerge.m`, three `.txt` files, and a `.dat` file) posted for Project 5 from the course website. You will also need the function `File2Cell` from `Insight` §11.1 (get the file from the course website). In `mailMerge.m` the main function and the first three subfunctions are for you to implement; the rest of the subfuctions have been implemented for your use.

Now we make sure that the downloaded `mailMerge` actually can be executed! Assuming that you've downloaded all the necessary files and they are all in the current directory, the following function call in the Command Window executes function `mailMerge` using the provided data files:

```
letters = mailMerge('sender.txt','recipients.txt','ltrTemplate.txt.','Zipcode.dat')
```

Read the function header and comments of `mailMerge` for details. After the function executes (almost instantaneously because the main function is practically empty), the resulting variable `letters` should be just an empty cell array. Now look at the code. Note that in each of the four (sub)functions that you need

to implement there is a statement that assigns to the return parameter an empty cell array or empty vector. These "dummy statements" in the file are called *stubs*; their sole purpose is to make the function body agree with the function header—create the return parameter—so that the function can execute. As you implement these functions you will *remove the stubs* and add the real code needed.

Now you know the given code actually executes, and your goal is to keep it that way as you add code! Test throughout program development.

1.2 A Person's Record

The record of a person consists of the person's first name, last name, mailing address, and ZIP+4 code. A person's record is represented in our data files as follows:

```
[first name] [last name]
[k, number of lines in the address]
[address line 1]
[address line 2]
...
[address line k]
[ZIP+4 code]
```

Here is an example of a person's record in one of our data files:

```
Ima Student
3
Pork Mountain Apartment
Room 2357
1234 Pork Hill Rd
03896
```

The name (first and last combined) of a person might include more than one space, e.g., the name `Solo Un Neung` includes two space characters. In this case, always assume that the first name ends at the first space. So the first name of `Solo Un Neung` is `Solo`, and the last name is `Un Neung`. Also, the ZIP+4 code may or may not contain the P.O. Box number. That is, both `03894` and `03894-4128` are valid ZIP+4 code for a record.

Observe that there is no city and state abbreviation in the record. This is because one can easily look up the city and state given the ZIP code, the first five characters of ZIP+4 code. Section 1.4 gives the details.

1.3 Sender and Recipients

We assume that there is only one sender. The information of the sender is specified as a person's record in a text file. See `sender.txt` for example. There may be zero or more recipients, each of which is also specified as a person's record. The recipients' records are grouped into another text file with the following format; see `recipients.txt` for example:

```
[n, number of recipients]
[record of 1st recipient]
[record of 2nd recipient]
...
[record of nth recipient]
```

We provide several subfunctions that process the sender and recipient data files for you, including these two:

```
function S = readSender(send_fname)
% Read sender's information from the given file having the specified format.
% send_fname: a string that gives the complete path to the file
% S: a structure containing the information of a person, the sender
% The function assumes there are no file format errors.

function RL = readRecipients(recp_fname)
% Read recipients' information from the given file having the specified format.
% recp_fname: a string that gives the complete path to the file
```

```
% RL: a structure array with component i being a struct containing the
%   information of the ith recipient
% The function assumes there are no file format errors.
```

The structures created in both functions contain the following fields:

- `first`: first name of the person (a string)
- `last`: last name of the person (a string)
- `addr`: address of the person, which is a string containing all the address lines except the ZIP information. The new line characters are inserted between the lines.
- `zip4`: ZIP+4 code of the person (a string)

In effect, the `readSender` and `readRecipients` functions are "make functions" that create for each person read from the data file a structure containing the person's data. Note that these functions assume there're no file format errors, i.e., the contents of the file follow the specifications above. **Do not modify the provided functions.** If you create your own data files, make sure they follow the specifications correctly.

The code in the given subfunctions are tedious but not difficult. Read the given code and make sure that you understand how the text is processed. The given code also gives you hints for code that you will need to write later. Specifically, note the following:

- Unlike simple arrays and cell arrays, a structure array cannot be initialized to a fixed size, including length zero. To create a structure array, simply assign to an indexed position of the array directly, e.g., `RL(k)=struct(...)` where `k` is a positive integer *without* initializing `RL` .
- `strfind` is a built-in function that finds one string within another. For example, `strfind('Single string','ing')` returns the vector `[2 11]`, where each vector component is the starting index in `'Single string'` where the string `'ing'` occurs.
- A single string may span multiple lines of text. Look at the code where the address lines are extracted. A single string (variable `address`) stores all the lines of an address. This is done by using the new line character `'\n'` and the function `sprintf` to build one single string out of all the lines.

Now that you understand the functions `readSender` and `readRecipients`, write code in the main function to call them. Then add some more code for testing—code that you will remove from the function once you're done testing. For example, suppose in the main function `mailMerge` you store the structure returned from `readSender` in variable `S`. Then you may want to try to display some of the field values in `S`. For example, the statements `disp(S.last)` and `disp(S.addr)` should display the last name and address of the sender. Similarly, `disp(RL(2).zip4)` should display the ZIP+4 code of the second recipient assuming that you store the returned structure array from `readRecipients` in variable `RL` (and there are at least two persons' information in the recipient data file).

1.4 ZIP Code Lookup

We define a Zipcode structure that has the following fields:

- `ZIP`: ZIP code (length-5 string)
- `city`: city name (a string with no *trailing* spaces)
- `state`: state abbreviation (length-2 string)
- `county`: county name (a string)

Implement the subfunction `MakeZipcode` as specified:

```
function Z = MakeZipcode(ZIP,city,state,county)
% Return a Zipcode structure Z such that
%   Z.ZIP is assigned the string in ZIP
%   Z.city is assigned the string in city
%   Z.state is assigned the string in state, and
%   Z.county is assigned the string in county
```

The function body can be as short as one statement! Don't be alarmed. (See *Insight* Chapter 10 or lecture notes for example.)

Our sender and recipient records do not include city and state names. Now we recover the city and state names using a ZIP code database so that those names will appear in a standard way (in mailing labels, for example). The text file `Zipcode.dat` has the following format:

- Columns 1-5: ZIP code
- Columns 7-33: city name
- Columns 35-36: state abbreviation
- Columns 38-[the length of the line]: county name

The following is one line in the database, where a ␣ indicates a space:

`14853␣Ithaca␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣NY␣Tompkins`

To *parse* the ZIP code database, implement the subfunction `readZIP`, which assumes that the format of the ZIP code file is as described above:

```
function C = readZIP(ZIP_fname)
% Read ZIP code information from the given file.
% ZIP_fname: a string that gives the complete path to the file
% C: a structure array with component i being a struct containing the
%    information of the ith ZIP code
```

Make effective use the functions `File2Cell` (given in *Insight* §11.1) and `MakeZipcode` in `readZIP`. Each structure in the structure array `C` is a Zipcode structure.

After implementing `readZIP`, call it from the main function (`mailMerge`). Again, add some test code in the main function to display the field values in a structure in the returned structure array. Note that `Zipcode.dat` is a big file and it takes a long time to process. Be patient, or better yet make your own, shorter, file for testing by copying just a few lines from the original file to a new file.

Now that you have the structure array containing information on the ZIP codes, it is possible to perform a lookup. Implement this subfunction:

```
function ZIPinfo = findZIP(C,ZIP)
% Return the information of the specified ZIP code
% C: a structure array containing the information of all the ZIP codes
% ZIP: the ZIP code (length-5 string) to be looked up
% ZIPinfo: a structure containing of the information of ZIP
% If the ZIP code is not found, ZIPinfo is the empty vector.
```

`ZIPinfo`, as well as every structure in `C`, is a Zipcode structure. A built-in function that you may want to use is `isempty`: `isempty(v)` returns 1 if `v` is an empty vector and returns 0 otherwise.

After implementing `findZIP`, test it by calling it from the main function (`mailMerge`). For example, the statement `SCity=findZIP(C,'14853')` where `C` is the array of Zipcode structures for all ZIP codes should result in a Zipcode structure `SCity` that has these field values:

```
    ZIP: '14853'
   city: 'Ithaca'
  state: 'NY'
county: 'Tompkins'
```

1.5 Merging

Here comes the fun part. First, remove the test code that you have written in `mailMerge`. To perform a mail merge, we need a template of the letter. A template consists of fixed text and placeholders. A placeholder begins with a `%` and ends with a `%`. The name of a placeholder is the string between the `%` symbols. Nevertheless, a percent sign can be generated using `%%`. See, e.g., the file `ltrTemplate.txt`. Placeholders never span across multiple lines, so there is always an even number of `%` symbols on each line. Each time you see a placeholder, you replace it with the information from the data source. Let's see an example.

1.5.1 Example

Suppose the sender file is

```
John Doe
2
Office of the University Commencement Events
B13 Day Hall, Cornell University
14853-2801
```

The recipient file is

```
2
Jackie Dough
2
Cornell University, Undergraduate Admissions
410 Thurston Avenue
14850-4321
Ima Student
3
Pork Mountain Apartment
Room 2357
1234 Pork Hill Rd
03896
```

And the template is

```
To %recipient% USA

%sender%

What a nice day in %sender_city%!  Today is %date%.
Don't make a mistake and address the letter to %recipient_first% %sender_last%.
```

Then `mailMerge` returns a cell array of length 2. The first cell contains a string that when displayed looks like this:

```
To Jackie Dough
Cornell University, Undergraduate Admissions
410 Thurston Avenue
Ithaca, NY 14850-4321 USA

John Doe
Office of the University Commencement Events
B13 Day Hall, Cornell University
Ithaca, NY 14853-2801

What a nice day in Ithaca!  Today is Tuesday, April 07, 2015.
Don't make a mistake and address the letter to Jackie Doe.
```

The second cell contains a string that when displayed looks like this:

```
To Ima Student
Pork Mountain Apartment
Room 2357
1234 Pork Hill Rd
Wolfeboro Falls, NH 03896 USA

John Doe
Office of the University Commencement Events
B13 Day Hall, Cornell University
Ithaca, NY 14853-2801

What a nice day in Ithaca!  Today is Tuesday, April 07, 2015.
Don't make a mistake and address the letter to Ima Doe.
```

1.5.2 Implement `mailMerge`

Implement the following function:

```
function letters = mailMerge(send_fname,recp_fname,tmpl_fname,ZIP_fname)
% Perform a mail merge given the data files
% send_fname: a string that gives the complete path to the sender file
% recp_fname: a string that gives the complete path to the recipient file
% tmpl_fname: a string that gives the complete path to the template file
% ZIP_fname: a string that gives the complete path to the ZIP code database
% letters: a cell array, each cell of which contains one string that is the
%   content of the letter to a recipient with valid ZIP code
% If the sender's ZIP code is invalid, letters is the empty cell array.
```

Here are all the defined placeholders.

- `sender`: the mailing address of the sender (including first name, last name, address, city, state, and ZIP+4; see an example in Section 1.5.1)
- `sender_first`: the first name of the sender
- `sender_last`: the last name of the sender
- `sender_city`: the city of the sender
- `sender_state`: the state of the sender
- `recipient`: the mailing address of the recipient
- `recipient_first`: the first name of the recipient
- `recipient_last`: the last name of the recipient
- `recipient_city`: the city of the recipient
- `recipient_state`: the state of the recipient
- `date`: today's date, which can be obtained from built-in functions `now` and `datestr`: `datestr(now,'dddd, mmmm dd, yyyy')` returns today's date as a string.

Replace any undefined placeholders with the string `'??'`. In addition, display the message

```
Undefined placeholder: [placeholder name].
```

on a separate line on the screen every time an undefined placeholder is encountered.

We process one line in the template at a time. It does not matter how long (or short) each line is after the replacements. When we are done, we have a finalized text which we call a *letter*, which is *one* string that spans multiple lines. Because we have a number of recipients, we generate one letter for each recipient. There are certain exceptions:

- If the ZIP code of the sender is not found in the database, then no letters are generated at all. In addition, display the message

  ```
  Sender ZIP code ([5-digit ZIP code]) does not exist.
  ```

  on a separate line on the screen.

- If the ZIP code of a recipient is not found in the database, then a letter for that particular recipient is not generated. In addition, display the message

  ```
  Recipient ZIP code ([5-digit ZIP code]) does not exist.
  ```

  on a separate line on the screen. This means that the length of cell array `letters` may be smaller than the number of recipients in the recipient data file.

Notes on the New Line Character

To generate a string that spans across multiple lines, use the built-in function `sprintf` effectively. The usage of `sprintf` is identical to that of `fprintf`, but instead of printing directly to the screen, `sprintf` returns the formatted string. For instance,

```
rep=sprintf('%s %s\n%s\n%s, %s %s',first,last,addr,city,state,zip4);
```

assign a string formatted for the address field to placeholder `rep`. Recall that the character `'\n'` is used to begin a new line.

1.6 Final Testing

The posted file `recipients.txt` contains the data for four recipients, one of which has an invalid ZIP code. If all your functions work correctly, there should be three letters generated only (cell array `letters` has length 3). For your own testing display the letters so that you can check the result visually. (E.g., type `letters{2}` in the Command Window to display the second letter.)

Feel free to come up with your own template! Have fun sending letters! Then submit your completed file `mailMerge.m` on CMS.